



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SAMULI REKONEN
APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSOR FOR
FUTURE RADIO INTEGRATED CIRCUITS

Master of Science Thesis

Examiner: Prof. Jarmo Takala
Examiner and subject approved by
the Faculty Council of the Faculty of
Computing and Electrical Engineering
2nd of May 2017

ABSTRACT

SAMULI REKONEN: Application-Specific Instruction-Set Processor for Future Radio Integrated Circuits
Tampere University of Technology
Master of Science Thesis, 42 pages
June 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Jarmo Takala

Keywords: application-specific instruction-set processor, licensed assisted access, hardware design, processor architecture

Licensed Assisted Access is a 3GPP specified feature, for using the unlicensed frequency band as a supplemental transmission medium to the licensed band. LAA uses clear channel assessment, for discovering the channel state and accessing the medium. LAA provides a contention based algorithm, featuring a conservative listen-before-talk scheme, and random back-off. This CCA scheme is thought to increase co-existence with existing technologies in the unlicensed band, namely, WLAN and Bluetooth.

Application-specific instruction-set processors can be tailored to fit most applications, and offer increased flexibility to hardware design through, programmable solutions. ASIP architecture is defined by the designer, while the ASIP tools provide retargetable compiler generation and automatic hardware description generation, for faster design exploration.

In this thesis, we explore the 3GPP LAA downlink requirements, and identify the key processing challenges as FFT, energy detection and carrier state maintenance. To design an efficient ASIP for LAA, we explore the different architectural choices we have available and arrive at a statically scheduled, multi-issue architecture. We evaluate different design approaches, and choose a Nokia internal ASIP design as the basis for our solution. We modify the design, to meet our requirements and conclude that the proposed solution should fit the LAA use case well.

TIIVISTELMÄ

SAMULI REKONEN: Sovelluskohtainen käskykantaprosessori tulevaisuuden radiomikropiireihin

Tampereen teknillinen yliopisto

Diplomityö, 42 sivua

Kesäkuu 2017

Tietotekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Pervasive Systems

Tarkastaja: professori Jarmo Takala

Avainsanat: sovelluskohtainen käskykantaprosessori, laitteistosuunnittelu, prosessoriarkkitehtuuri, järjestelmäpiiri

”Licenced Assisted Access” -ominaisuus (LAA) on 3GPP:n määrittelemä menetelmä, joka pyrkii mahdollistamaan vapaiden radiotaajuusalueiden käytön, lisensoituja taajuuksia täydentävänä lähetykskaistana. LAA tulkitsee lähetykskanavien tilaa käyttäen CCA algoritmi. CCA on konservatiivinen kuuntele-ennen-lähetyks algoritmi, jossa käytetään satunnaistusta törmäysten vähentämiseksi. Koska vapailla taajuuksalueilla on jo käyttäjiä, tärkeimpinä langattomat lähiverkot ja Bluetooth, yhdessä näiden teknologioiden kanssa on yksi LAA:n tärkeimmistä vaatimuksista.

Sovelluskohtaisia käskykantaprosessoreita voidaan räätälöidä lähes jokaiseen sovellukseen ja ne tarjoavat joustavia ratkaisuja laitteistosuunnitteluun ohjelmoitavien ratkaisujen avulla. Sovelluskohtaisten käskykantaprosessoreiden arkkitehtuurin määrittelee itse laitteiston suunnittelija, jonka avulla prosessoreiden suunnitteluun tarkoitetut työkalut generoivat uudelleen kohdistettavan sovelluskääntäjän. Tämä helpottaa laitteistokehitystä vähentämällä suunnittelijan työtaakkaa ja nopeuttamalla suunnittelukierroksia.

Tässä diplomityössä tarkastelemme 3GPP LAA -määrittelyn vaatimuksia, joista identifioimme tärkeimmät prosessointialgoritmit, kuten nopea Fourier -muunnos, kaistan energian havainnointi ja kanavan tilakoneen ylläpito. Tutkimme myös mahdollisia prosessoriarkkitehtuureja, joilla voisimme tehokkaasti toteuttaa kyseiset laskennalliset vaatimukset ja havaitsemme staattisesti aikataulutetun rinnakkaisarkkitehtuurin soveltuvan vaatimuksiimme hyvin. Valitsemme laitteistokehityksen lähtökohdaksi Nokialla suunnitellun prosessorin, jota muokkaamalla päädyttiin ratkaisuun, joka täyttää asettamamme suunnitteluvaatimukset ja soveltuu LAA -sovellukseen erinomaisesti.

PREFACE

I would like to thank everyone at Nokia who provided support for this thesis, especially Eric Borghs who supplied the initial processor design. Also, thanks to professor Jarmo Takala for guidance during the writing process, and accommodating my busy schedule.

Finally, a special thank you to my friends, family and NBA basketball for keeping me sane throughout this process.

In Tampere, Finland, 24th of May 2017

Samuli Rekonen

TABLE OF CONTENTS

1.	INTRODUCTION	1
2.	LICENSED ASSISTED ACCESS.....	3
2.1	Clear Channel Assessment.....	4
2.1.1	Multi-Carrier Listen-Before-Talk	7
2.2	Dynamic Frequency Selection	8
2.3	Processing Algorithms	9
2.3.1	Fast Fourier Transform	9
2.3.2	Energy Detection.....	10
2.4	Beyond 3GPP Release 13.....	11
2.4.1	Uplink Licensed Assisted Access	12
2.4.2	Access with 32 Carriers	12
3.	APPLICATION-SPECIFIC INSTRUCTION SET PROCESSOR	13
3.1	Instruction-Level Parallelism	14
3.1.1	Structural Hazards.....	16
3.1.2	Data Dependencies and Hazards.....	17
3.1.3	Control Dependences and Hazards.	18
3.1.4	Dynamic vs. Static Scheduling	19
3.2	Multiple-Issue Architectures	21
3.2.1	Very Long Instruction Word Architecture.....	22
3.2.2	Transport Triggered Architecture	23
3.3	Data-Level Parallelism.....	24
3.3.1	Vector Processors.....	25
3.4	ASIP Tools and Methodologies	26
3.4.1	LISA.....	26
3.4.2	nML.....	26
3.4.3	ASIP Designer by Synopsys	27
3.4.4	TTA-Based Co-Design Environment.....	28
4.	IMPLEMENTATION	30
4.1	Processing Steps.....	30
4.2	Implementation Approaches	32
4.2.1	ASIP with FFT Accelerator	32
4.2.2	ASIP-Only Solution	33
4.3	ASIP Implementation	33
4.3.1	Design Exploration	34
4.3.2	Architecture.....	35
5.	VERIFICATION AND ANALYSIS	37
5.1	Verification.....	37
5.2	Synthesis.....	39
5.3	Further Development.....	40
6.	CONCLUSIONS.....	42

REFERENCES.....43

SYMBOLS AND ABBREVIATIONS

3GPP	3 rd Generation Partnership Program
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-Set Processor
CCA	Clear Channel Assessment
CR	Cognitive Radio
CSI	Channel-State Information
CSMA/CA	Channel Sense Multiple Access and Collision Avoidance
CWS	Contention Window Size
DFS	Dynamic Frequency Selection
DFT	Discrete Fourier Transform
DL	Downlink
DLP	Data-Level Parallelism
DMA	Direct Memory Access
FFT	Fast Fourier Transform
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HARQ	Hybrid Automatic Repeat Request
IDE	Integrated Development Environment
ILP	Instruction-Level Parallelism
IP	Intellectual Property block
ISA	Instruction-Set Architecture
LAA	Licensed Assisted Access
LBT	Listen-Before-Talk
LTE	Long Term Evolution
MIMD	Multiple Instructions Multiple Data
MIMO	Multiple-Input Multiple-Output
NACK	Negative Acknowledgement
NCTL	Network Controller
NOP	No Operation
NRE	Non-recurring Engineering Cost
PDCCH	Physical Downlink Control Channel
PUCCH	Physical Uplink Control Channel
RAW	Read-After-Write
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
UE	User Equipment
UL	Uplink
UVM	Universal Verification Methodology
TCE	TTA-Based Co-Design Environment
TTA	Transport Triggered Architecture
TUT	Tampere University of Technology
VLIW	Very Long Instruction Word architecture

WAR	Write-After-Read
WAW	Write-After-Write
WLAN	Wireless Local Area Network

1. INTRODUCTION

With a number of devices accessing the Internet over wireless networks almost exponentially increasing, the licensed frequency spectrum is getting more and more congested, while our performance expectations have been growing with the advances in technology. Applications such as video streaming and online gaming are commonplace applications that are increasingly used in mobile devices. These applications require high data rates with low end-to-end latency. Since one of the easiest methods to increase data rate is increasing the available bandwidth, the next evolution of mobile network standards, 5G, aims to do just that. With the licensed spectrum already congested, the increased bandwidth needs to come from the unlicensed frequency spectrum.

The proposal for 5G standard is to utilize the unlicensed band of under 5GHz, with a feature called Licensed Assisted Access (LAA). These frequency bands are not without users, so fair co-existence with the existing systems is key. To accommodate this, the standard features an algorithm that aims to reduce collisions, while keeping the channel occupied as much as possible. This algorithm is called LAA Clear Channel Assessment (CCA), it is contention based, and features randomization for increased fairness.

To utilize the LAA feature, we need hardware capable of performing the CCA algorithm, and since this feature is brand new, we must design it ourselves. Over the years, hardware designs have been getting increasingly complex, while increases in designer's productivity has not been keeping up, which has lead us to look for more productive solutions for hardware design. Application-specific instruction-set processors (ASIPs) are looking to close that gap in productivity. ASIPs offer programmable solutions, that are designed at a higher abstraction level than manual VHDL and Verilog coding, and the toolsets can be used to automatically generate the hardware description as well as a compiler for that specific processor. ASIP architecture can be tailored for most applications, and offer a powerful way to potentially increase the product lifetime, with programmable hardware.

The purpose of this thesis is to study the LAA specification, and its evolution, and implement an integrated circuit solution for Nokia, that can perform CCA in a cognitive radio chip. To do this we will need to understand the application area, and its requirements, as well as the design methodology and the impact of our architectural decisions, which we aim to do in this thesis.

The thesis is structured as follows, Chapter 2 introduces the LAA feature and its requirements, and the algorithms we need to execute in our processor. Chapter 3, takes a deep

dive into processor architecture, so that we are well informed when defining the processor. Chapter 4, gathers our design criteria, and we explore the design space for potential solutions. Finally, in Chapter 5, we describe how the design was verified, and analyze the results.

2. LICENSED ASSISTED ACCESS

Mobile communication systems have evolved from only supporting analog voice transmissions to delivering thousands of different applications to billions of users. With the world moving towards a more networked society, with the emergence of household and industrial Internet-of-Things applications, there is a clear need for the next generation of wireless standards to support a drastic increase in users, data rate and reduced latency. [1]

With the increasing data rate demands, there is a substantial increase in demand for the licensed frequency spectrum. Some of the techniques to increase network capacity in Long Term Evolution (LTE) systems include: higher order modulation schemes, advanced multi-input multi-output (MIMO) antenna technologies [2]. However, the licensed cellular spectrum is already congested. Since the licensed band is unable to accommodate the large jump in data rate, the 3rd Generation Partnership Project (3GPP) specification focus for 5G has been shifted to the use of unlicensed band of over 500MHz in LTE systems [3], Figure 1 shows the frequency spectrums that have proposed study items in 5G scope. These studies have been focused on the use of frequency bands under 5GHz and 3GPP Release 13 describes a procedure to enable downlink (DL) operation of LTE systems in unlicensed band [2]. Release 14 will include uplink (UL) specification but, for the purpose of this thesis we will focus on only DL operation.

Since 3GPP considers unlicensed spectrum as supplemental to licensed spectrum, this feature is called licensed-assisted-access to unlicensed spectrum, hence the feature is often referred to as LAA. One of the key considerations for using the unlicensed spectrum is to ensure fair coexistence with the incumbent systems such as Bluetooth, Zigbee and wireless local area networks (WLANs). The latter is the principal focus of LAA standardization effort with IEEE and Wi-Fi Alliance who define the standards for WLAN systems. [2]

The standardization of LAA for Release 13 was conducted in two phases; first a study item phase, followed by the working item phase [4]. The goal of the study item phase was to study the feasibility of LTE enhancement to enable LAA operation in unlicensed spectrum while coexisting with the incumbent systems and fulfilling the regulatory requirements. The study item phase concluded that LAA can fairly coexist with WLANs and other unlicensed spectrum systems, if an appropriate channel access scheme is adopted [2]. The proposed access scheme is a conservative listen-before-talk (LBT) scheme [4], which we will take a thorough look at later.

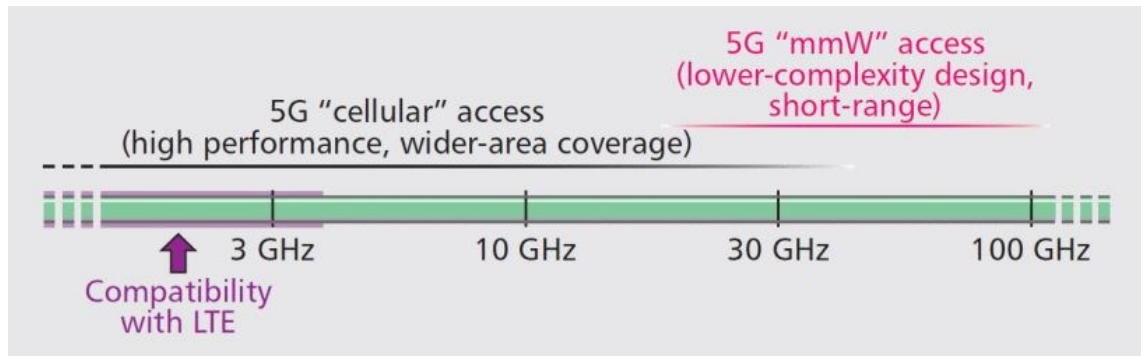


Figure 1. Spectrum range to be considered for 5G wireless access [1].

The work item phases main objective was to specify LTE enhancements for operation in unlicensed spectrum, under the design criteria of a single global solution framework, fair coexistence between LAA and WLAN networks and fair coexistence with other LAA networks. The detailed objectives for the work item were to specify: channel access framework, including clear channel assessment; discontinuous transmission with limited maximum transmission duration; user equipment (UE) support for carrier selection; UE support for radio resource management measurements; time and frequency synchronization; and channel-state information (CSI) measurement. The LAA work item was completed in late 2015. [2]

The usage of LTE in unlicensed spectrum is a fundamental paradigm shift, since LTE physical channels have largely been designed on the basis of uninterrupted operation on licensed carriers [5]. Now with LAA framework the primary use case is a supplemental transmission medium using carrier aggregation and discontinuous transmission. Since the unlicensed carrier is shared by multiple systems, collisions are unavoidable. This is why LAA can never match the licensed carrier in terms of mobility, reliability and quality of service. [2]

2.1 Clear Channel Assessment

In Japan, Europe, and the United States, unlicensed spectrum between 455MHz and 555MHz is currently available for use in the 5GHz band. This unlicensed band can be divided into multiple carriers of 20MHz each. Therefore, the first step to ensure fair coexistence for LAA nodes is to select a carrier with low interference for operation. However, carrier selection by itself is not a sufficient method to ensure collision free operation. Sharing of unlicensed carriers between different technologies, like WLAN and Bluetooth, is inevitable. [5]

Since static carrier selection is not enough for fair coexistence, LAA incorporates similar channel access techniques as WLAN, mainly LBT and random-back-off. These techniques are fairly similar to IEEE 802.11n/ac standard, which defines a contention-based channel access method for WLANs called: carrier sense multiple access with collision avoidance (CSMA/CA). LBT is a procedure, where radio transmitters first sense the physical medium and transmit only if the medium is sensed to be idle, which is also called clear channel assessment (CCA). CCA utilizes at least energy detection to determine if there are signals being transmitted on the channel. The other part of LAA CCA is random-back-off, which is designed for collision avoidance. After determining that the channel is idle multiple nodes might wish to transmit at once, so each node will wait for a random number of idle periods on the channel before starting transmission [2]. The size for the generated random number for back-off clearly has a big impact on the efficiency of the medium; if the number tends to be big the channel will be idle more and on the other hand small numbers will increase the probability of collisions.

Figure 2 shows the recommended CCA method for LAA operation as described in 3GPP Rel 13 [6]. The following steps illustrate the CCA state machine algorithm.

1. If the node wishes to transmit on the selected carrier, first it must sense that the medium is idle for initial defer period T_d before moving to the step number 2.
2. The node will pick a random number uniformly distributed between 0 and the maximum value of the Contention Window Size (CWS) denoted as q in Figure 2. Then move to the step number 3.
3. The node must sense the channel is idle for a slot duration T_{sl} , denoted as D in Figure 2 before moving to the step number 4.
4. If $N = 0$, stop CCA and transmit on the carrier. If $N \neq 0$ the node may choose to decrement its N counter or not before going back to the step number 3. The decision to decrement the counter or not should be based on a random procedure.

In the step 3 above, a slot duration T_{sl} is considered to be idle if the node senses the channel during the slot duration, and the power detected by the node for at least 4 within the slot duration is less than the energy detection threshold. Otherwise, the slot duration T_{sl} is considered to be busy. [6]

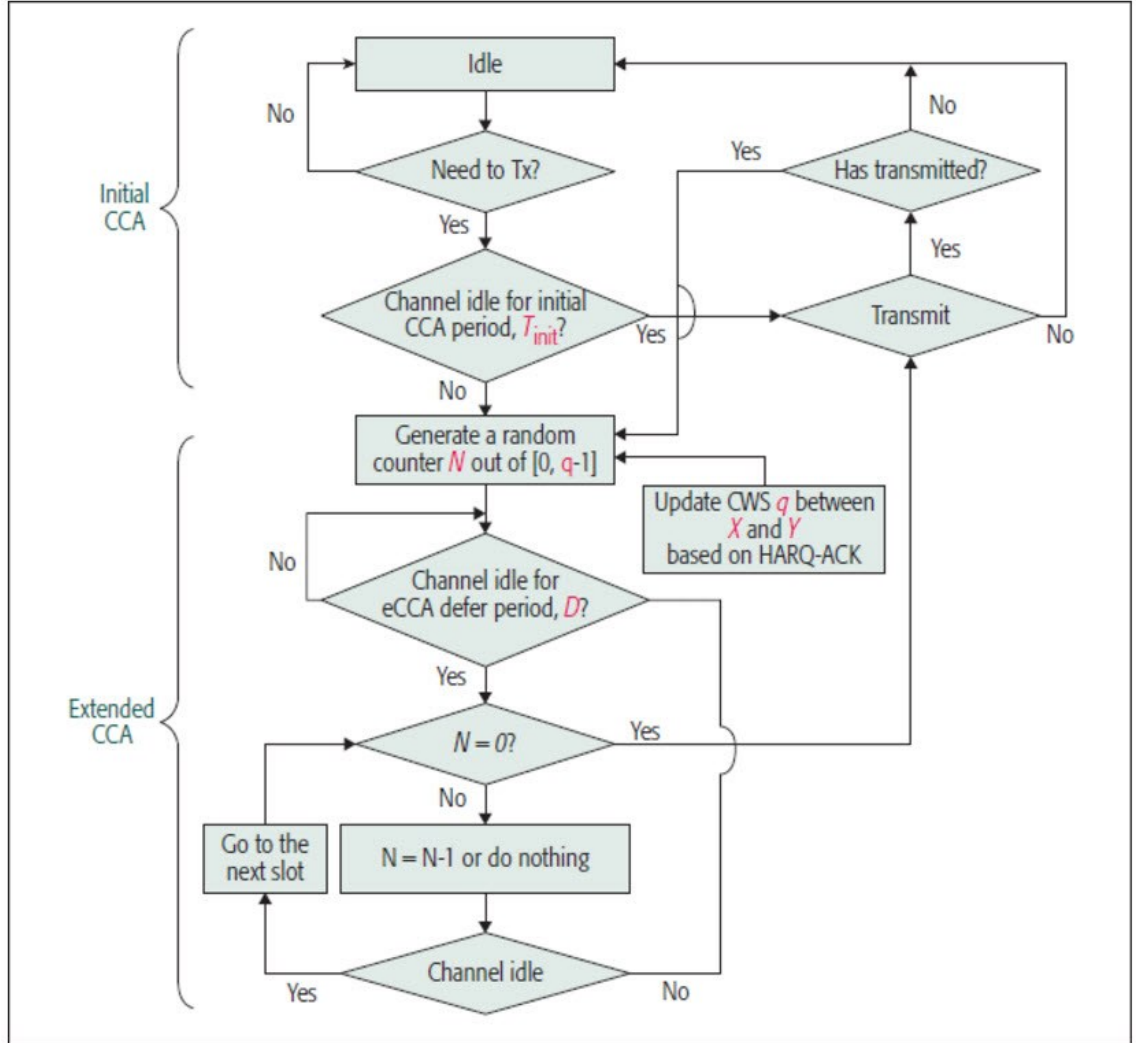


Figure 2. Flowchart of the recommended DL LBT procedure by 3GPP [2].

3GPP defines the defer and slot durations as; defer duration T_d consists of duration $T_f = 16\mu\text{s}$ immediately followed by m_p consecutive slot durations, where each slot duration is $T_{sl} = 9\mu\text{s}$, and T_f includes an idle slot duration T_{sl} at start of T_f . There is also a maximum transmit period $T_{mcot,p}$, this parameter, as well as CWS and m_p , are defined by the Channel Access Priority Class [6]. The values these parameters may obtain are listed in Table 1.

Table 1. *Channel Access Priority Classes and related parameters [6].*

Channel Access Priority Class	m_p	$T_{mcot,p}$ (ms)	CWS
1	1	2	{3, 7}
2	1	3	{7, 15}
3	3	8 or 10	{15, 31, 63}
4	7	8 or 10	{15, 31, 63, 127, 255, 511, 1023}

The CWS is adjusted based on feedback from user equipment. If the Hybrid Automatic Repeat Request (HARQ) feedback for the most recent DL transmission burst had 80 percent or more decoding errors i.e. negative acknowledgments (NACKs) then the CWS is double for the next CCA. Otherwise the CWS is reset to the minimum value. [5]

Earlier we mentioned that the channel occupancy is determined by the received signal level that we compare to an energy detection threshold value. This value determines the level of sensitivity for detecting ongoing transmissions. If the absence of any other technology sharing the carrier cannot be guaranteed on a long term basis (e.g. by level of regulation) then the maximum energy detection threshold is

$$TH = \max \left\{ \begin{array}{l} -72 + 10 * \log_{10} \left(\frac{BW}{20MHz} \right) dBm, \\ \min \left\{ T_{max}, T_{max} - T_A - 10dB + (P_H - P_{TX}) \right\} \end{array} \right\}, \quad (1)$$

where P_H is a reference power equaling 23 dBm, P_{TX} is the configured maximum transmit power for the carrier in dBm, and

$$T_{max}(dBm) = 10 * \log_{10} \left(3.16228 * 10^{-8} \left(\frac{mW}{MHz} \right) * BW(MHz) \right). \quad (2)$$

where BW denotes the single carrier bandwidth in MHz. [6]

2.1.1 Multi-Carrier Listen-Before-Talk

In the previous chapter, we only considered LAA operation on a single carrier but simultaneous operation on multiple unlicensed carriers is a key technique for maximizing the data delivered during a transmission opportunity. For WLANs IEEE 802.11ac supports transmission bandwidth of up to 160 MHz, which would span eight contiguous 20 MHz unlicensed channels in the 5 GHz band. LAA multi-carrier design should continue to adhere to the principle of fair coexistence with WLAN design, while being able to quickly detect transmission opportunities across multiple carriers. [5]

The scheme used by WLANs could be described as a hierarchical channel bonding scheme where 20 MHz sub-carriers are combined in a non-overlapping manner. One of these contiguous sub-carriers is designated as a primary channel on which a complete

random back-off cycle is performed, while the others are designated as secondary carriers. Counting down of the random back-off counter is based only on the outcome of clear channel assessments on the primary carrier. On the secondary carriers, only a quick CCA check is performed before the potential start of transmission. Upon expiration of the back-off counter on the primary carrier, the overall transmission bandwidth; i.e. 20MHz, 40 MHz, 80MHz, or 160MHz, is determined by the results of the secondary CCA checks. The energy detection thresholds for secondary carriers are generally higher than for the primary carrier and scale up with increasing channel bandwidth. [5]

Thus, 3GPP Release 13 defines two types of access schemes for LAA multi-carrier LBT:

- **Type A: Parallel Random Back-off Channels:** Each carrier selected for transmission must evaluate the carrier state by performing the full CCA procedure we defined earlier in chapter 2.1. Each carrier also needs to maintain its own random back-off counter.
- **Type B: Single Random Back-off Channel:** Similar to WLAN, only one full-fledged CCA procedure must be completed on the primary carrier while the secondary carriers will only perform a short CCA. The sensing period for secondary cells needs to be at least 25 μ s. [6]

2.2 Dynamic Frequency Selection

Radar detection function of Dynamic Frequency Selection (DFS) is a regulatory requirement for discovery and avoidance of frequencies used by radar systems. This feature is not defined in 3GPP Release 13 since their previous study concluded that the feature does not require further specification and is an implementation issue. [3] Let's now take a look at what additional requirements this might impose on our system.

The study concluded that according to the FCC, devices operating in the 5,25-5,35 GHz and 5,47-5,725 GHz frequency bands must employ a DFS radar detection mechanism to detect the presence of radar systems. These devices must sense for the presence of radar systems at 100% of its emission bandwidth. The presence of radar systems is detected by an energy detection threshold. If a radar system is detected on the frequency band the device must move to a different channel, and may only start using the new channel if no radar signal is detected within 60 seconds. [3]

After a radar signal is detected on the channel, all transmissions need to cease on the channel within 10 seconds. Normal traffic can be continued for a maximum of 200 ms after detection of the radar signal, intermittent management and control signals can be sent during the remaining time to facilitate vacating the operating channel. [3]

Since our system will only be responsible for the CCA part of LAA, for DFS we need only to detect the presence of radar systems on the channel and inform the transmission

system. This means apart from, carrying out energy detection and comparing it to the energy detection threshold defined in chapter 2.1, we must compare the energy detection results to this new DFS threshold. Now that we have a high-level view of how our system should work, let us discuss these processing steps in more detail.

2.3 Processing Algorithms

In order to perform CCA, we must determine the transmission medium's state by performing energy detection. For this, we need to sample the transmission medium to obtain the instantaneous in-phase (I) and quadrature (Q) components of the signal, which can be represented as a single complex value, where the real part represents I and the complex part Q. These IQ-data samples are captured in time-domain, but we would like to perform the energy detection by analyzing the frequency spectrum of the signal in a given discrete time interval. For this, we need to transform the IQ-samples from time-domain to frequency-domain.

2.3.1 Fast Fourier Transform

Transforming signals from time-domain to frequency-domain can be done using Discrete Fourier Transform (DFT) algorithm, which finds the spectrum of a periodic discrete-time signal with period N . The spectral component is obtained by

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad 0 \leq k < N, \quad (3)$$

where W_N is the twiddle factor, which equals the N th root of unity and is given by

$$W_N = e^{-\frac{2j\pi}{N}}; \quad j = \sqrt{-1}. \quad (4)$$

If we evaluate the complexity of algorithm (3), we see that it requires $(N - 1)^2$ complex-valued multiplications and $N(N - 1)$ additions. With big O notation, this gives us a complexity of $O(n^2)$. This algorithm is then quite heavy on complexity, but fortunately for us there is a more efficient way to perform this same transform, commonly called the Fast Fourier Transform (FFT). [7]

Introduction of the FFT to the world is often credited to Cooley and Tukey [8], for their paper in 1965 called “An Algorithm for the Machine Calculation of Complex Fourier Series”. In the paper, they introduce a way to reduce the complexity of the DFT to $O(n \log n)$, by decimating the original sequence in to smaller sequences. This reduction in complexity is quite significant when the number of samples N increases.

One common variation of the FFT algorithm is the decimation-in-time FFT, which breaks the sum in equation (3) into even- and odd-numbered pairs. These even and odd sequences are given by

$$x_0(n) = x(2n) \quad n = 0, 1, \dots, \left(\frac{N}{2}\right) - 1, \quad (5)$$

$$x_1(n) = x(2n + 1) \quad n = 0, 1, \dots, \left(\frac{N}{2}\right) - 1, \quad (6)$$

where x_0 denotes the even sequence and x_1 the odd sequence. Now the sum in (3) can be written as

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_N^{2(n+1)k} \quad 0 \leq k < N. \quad (7)$$

We can write W_N^2 as

$$W_N^2 = (e^{-\frac{2j\pi}{N}})^2 = e^{-\frac{2j\pi}{N/2}} = W_{N/2}. \quad (8)$$

Finally, we can write equation (7) as

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x_0(n) W_{N/2}^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_{N/2}^{nk} \quad 0 \leq k < \frac{N}{2}, \quad (9)$$

which simplifies to

$$X(k) = X_0(k) + W_N^k X_1(k) \quad 0 \leq k < \frac{N}{2}, \quad (10)$$

where $X_0(k)$ and $X_1(k)$ are the $N/2$ -point DFTs of $x_0(n)$ and $x_1(n)$ respectively. Here, we need to notice that equation in (10) $X_0(k)$ and $X_1(k)$ are defined for $0 \leq k < \frac{N}{2}$, while in our original equation (3) $X(k)$ is defined for $0 \leq k < N$. Since $X_0(k)$ and $X_1(k)$ are periodic with a period of $N/2$, we can express equation (10) as

$$X\left(k + \frac{N}{2}\right) = X_0(k) - W_N^k X_1(k) \quad 0 \leq k < \frac{N}{2}. \quad (11)$$

Here (10) and (11) are commonly referred to as the *butterfly* operations. [7]

2.3.2 Energy Detection

As noted earlier, the CCA algorithm uses energy detection followed by comparing it to the energy detection threshold, defined in (1) and (2), to determine whether the carrier is free or not. A widely used algorithm for energy detection was introduced by Urkowitz in 1967 [9]: “by using Shannon’s sampling theorem, one can show that the energy in a finite time interval can be described as a sum of the square over a number of statistically independent Gaussian variates if the noise input is Gaussian and has a flat spectral density over a limited bandwidth.” So our energy detector will compare the signal to two hypotheses:

1. H_0 : Nobody occupies the channel.
2. H_1 : The channel is occupied by signal with noise.

The received power T is given by:

$$T = \frac{1}{N} \sum_{n=1}^N |x(n)|^2 = \frac{1}{N} \sum_{n=1}^N |w(n)|^2 : H_0,$$

$$T = \frac{1}{N} \sum_{n=1}^N |x(n)|^2 = \frac{1}{N} \sum_{n=1}^N |hs(n) + w(n)|^2 : H_1, \quad (12)$$

where $x(n)$ denotes the n -th received sample, $w(n)$ is additive white Gaussian noise, $s(n)$ is a signal which occupies the observed carrier, h is a channel coefficient, and the number of samples is denoted by N [10].

2.4 Beyond 3GPP Release 13

As stated previously, for this thesis we want to limit the use case for our design to the DL LAA transmission that we discussed earlier. But since the system we are designing would ideally be integrated into a baseband or radio Application-Specific Integrated Circuit (ASIC), that might have a lifetime measured in tens of years, we should also consider additional features that might arise from now, until all the specifications for 5G are frozen. Figure 3 illustrates ITU-Rs vision for 5G specification process and from it we can see that the standardization activities will continue until 2020. High frequency 5G showcases are expected for Tokyo 2020 Olympics [11].



Figure 3. ITU-R time timeline for the work on "IMT-2020" specifications [1].

2.4.1 Uplink Licensed Assisted Access

In LTE, the UL access is scheduled. The UE is not allowed to transmit data unless it is granted resources to do so. The UE must notify when data is available for transmissions using the Physical Uplink Control Channel (PUCCH) and indicating that it needs UL access. Then it will wait to receive a UL grant message from the Physical Downlink Control Channel (PDCCH), which contains the information needed by the UE to perform the UL transmission. [12]

LAA UL follows the same principle to grant access for UL transmission. As required on unlicensed bands, the UE is required to perform CCA checks using energy detection before transmitting. This puts LTE systems at a disadvantage in the unlicensed band, since WLAN systems are autonomous and can asynchronously transmit at any free slot, not restricted by grants for transmissions and not required to have explicit permission to transmit. This allows a WLAN node to more aggressively contend for the channel and acquire transmission access. A way to combat this issue will be described in 3GPP Release 14, but some of the challenges inherent in the scheduled UL concept were left open for future enhancements. [12]

2.4.2 Access with 32 Carriers

Wideband transmissions are a key feature for enabling high user data rates, and this is especially true as we evolve towards 5G. IEEE 802.11ac currently supports transmission bandwidths of up to 160 MHz, and further improvements are expected in 802.11ax. Therefore, LAA should be enhanced to support system bandwidths similar to 802.11ac in unlicensed spectrum. A separate 3GPP work item specified aggregation of up to 16 or 32 carriers, which is a natural candidate for application to LAA. With 32 aggregated carriers, LAA will be able to support transmissions of 640 MHz to a single UE. This feature would impact a number of physical layer aspects, such as the need to support PUCCH on LAA to reduce control overhead and enhancements in scheduling with such a large number of available carriers. [5]

3. APPLICATION-SPECIFIC INSTRUCTION SET PROCESSOR

ASIC design has been increasing in complexity with the adaptation of smaller and smaller technology geometries, and design tools are finding it difficult to handle the complexity and electrical design challenges posed by each new technology generation. Consequence of this is lowered design productivity despite increasingly expensive design tool licensing fees. [13] The increasingly high nonrecurring engineering (NRE) costs of migrating to smaller ASIC technologies only amortize for very large volumes [14].

Early System-on-chip (SoC) designs were of low complexity, typically comprising one or two programmable processing units with custom hardware, peripherals and memory [14]. An alternative implementation style, that has been emerging since the early 2000s, is the use of programmable solutions. For a hardware developer the programmability of these devices enables a larger volume of applications being mapped to the same device, and possibly even newer product generations using the same device. Programmable solutions are also cheaper to debug, since writing and debugging software is much cheaper than designing, debugging and manufacturing working hardware. This reduction in debug complexity could lead to faster time-to-market, which is increasingly important in realizing commercially successful designs. While programmability provides flexibility and reduces debugging effort, it comes at a significant cost in performance and power consumption. [13] Thus, custom logic blocks, or semiconductor intellectual property cores (IP cores), will continue to tackle such requirements as very high performance and ultra low power consumption [14].

The optimal solution would clearly offer the flexibility of a programmable solution, while not making too many sacrifices in terms of area, performance and power. Application-specific instruction set processors (ASIPs) try to achieve this by providing specific hardware features for the targeted application. The ASIP instruction set and architecture can be derived using an iterative design space exploration process. Which is a feedback driven process, where each design requirement is evaluated with clearly defined metrics. The process should be an iterative loop where you define the requirements, develop the architectural model of the ASIP and build a software environment consisting of a compiler to map the application to the architecture, and simulators to quantify the quality of the result [13]. After which you run your compiled code with the instruction set simulator to see if the ASIP architecture and software fulfill their requirements, if not keep iterating until they do and you can generate the register transfer level (RTL) description of the ASIP. Figure 4 illustrates this process.

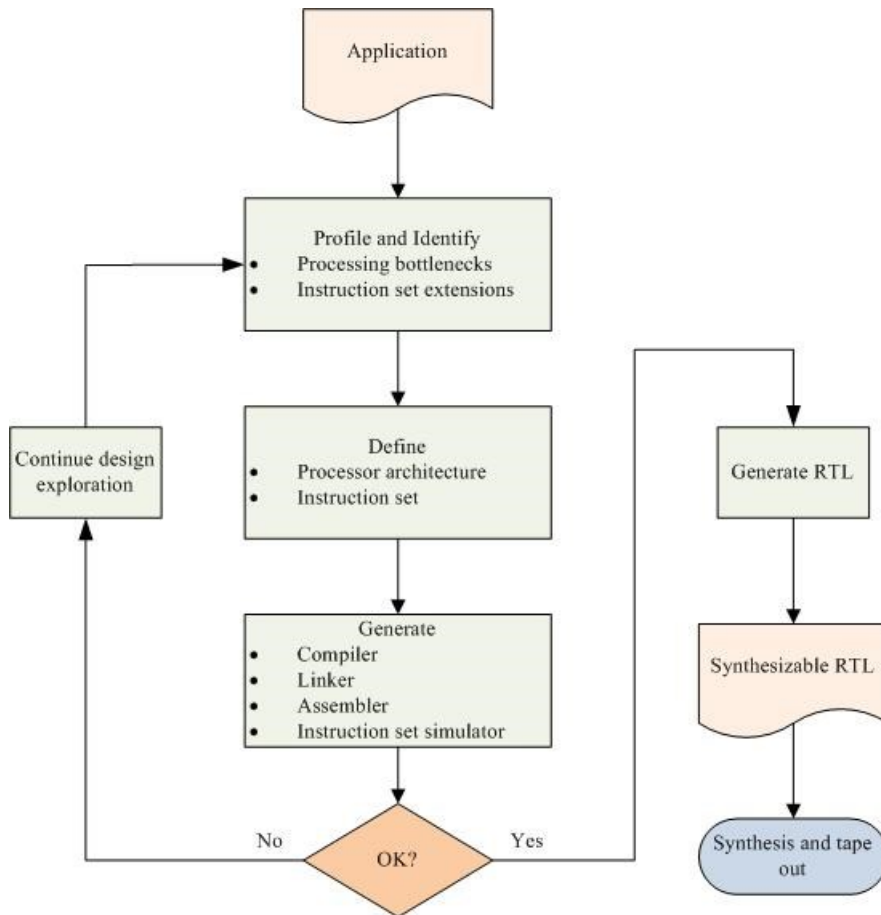


Figure 4. *A simplified design flow for an ASIP platform.*

Clearly the architecture, instruction set, and compiler that efficiently uses the devices resources, are determining factors for the ASIP to meet our area, performance and power requirements. Since most ASIP design tools, like TCE [15] and ASIP Designer [16], offer automatic targeted C/C++ compiler generation for the ASIP at hand, we do not need to look into creating our own. Instead, in this chapter, we will explore the architectural choices and challenges, such as parallelism and multiple-issue architectures.

3.1 Instruction-Level Parallelism

Basically every processor since about 1985 use a technique called pipelining to overlap the execution of instructions and improve performance. This overlap in instructions is called instruction-level parallelism (ILP), it takes advantage of parallelism that exists among the actions needed to execute an instruction. In a processor pipeline, each stage in the pipeline completes a part of an instruction. These stages are connected one to the next to form a pipe: instructions enter at one end, progress through the different stages, and exits at the other end. Since the pipe stages are linked, all the stages must be ready to

proceed at the same time. The time required to move an instruction from one stage to the next is called a processor cycle, this processor cycle is ideally one clock cycle. [17]

Pipelining reduces the average execution time per instruction, or in other words throughput since the execution time of any single instruction does not decrease. The reduction in average execution time arises from keeping each pipeline stage busy with part of one instruction. [17] A simplified pipeline of a Reduced Instruction Set Computer (RISC) is shown in Figure 5. From this figure, we can see the 5 stages of the pipeline filled in different points in time when there is a new instruction fed to the pipeline on every clock cycle. All the instructions take 5 clock cycles from start to finish but, after the first instruction completes on clock cycle number 5 we are completing a new instruction on each following clock cycle.

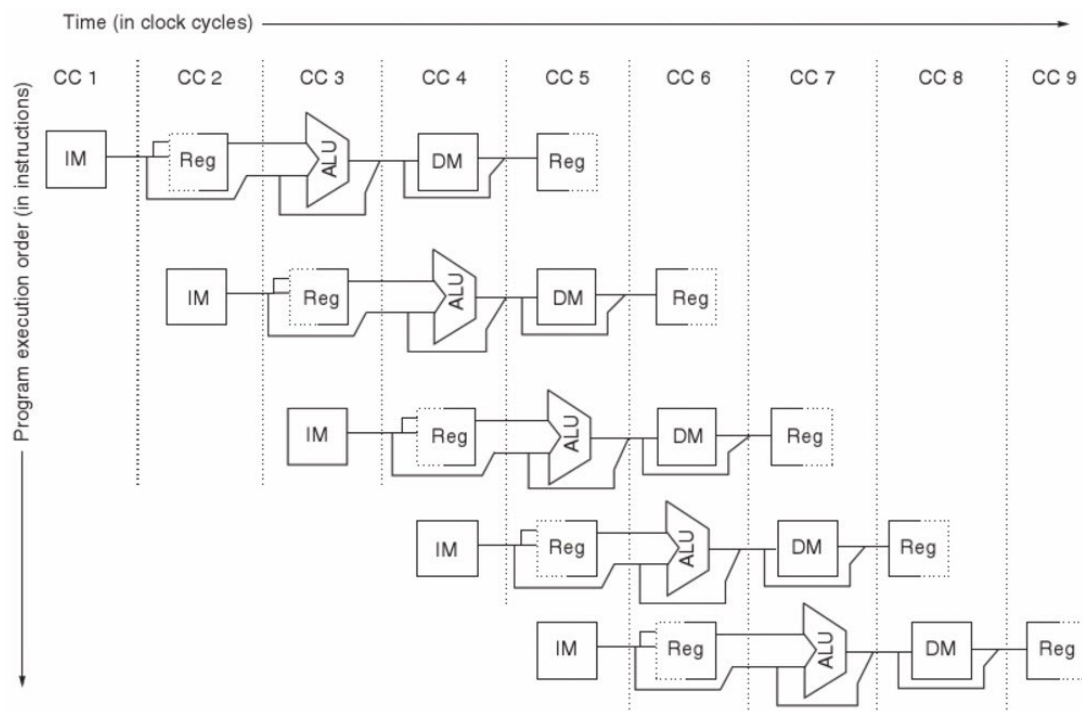


Figure 5. Pipeline of a RISC processor [17].

The example provided above is the ideal case, everything works as expected and our pipeline keeps completing new instructions on every clock cycle without a hitch. But since most software is sequential in nature and full of conditional jumps, such as if-else structures in C, our current ideal pipeline would not work in these cases. For example, say we have the following assembly style instructions in sequence:

```
ADD R0, R1, R2;
ADD R3, R0, R4;
```

here the first instruction takes the values in registers *R1* and *R2*, adds them together, and stores the results in register *R0*. Then the following instruction adds the values in registers

$R0$ and $R4$ together, and stores the result in $R3$. In a unpipelined processor nothing out of the ordinary happens, but if we again look at figure 5 for our pipelined model: the first instruction stores its result to register $R0$ on clock cycle 5, while the second instruction loads the value of register $R0$ on clock cycle 3. This will result in the second instruction loading the old value of $R0$ and not the value computed in the previous instruction. This is commonly referred to as a data dependency. And is one type of hazard that must be addressed for correct and efficient pipelined processor design. Hazards can generally be categorized in three classes:

1. *Structural hazards* are resource conflicts where multiple instructions try to access the same hardware unit on the same clock cycle.
2. *Data hazards* when an instruction depends on the results of a previous instruction in a way that it is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* are encountered when pipelining conditional jump or branch instructions.

Hazards can make it necessary to stall the pipeline, also known as bubbles in the pipeline. When the processor is stalled, no new instructions are issued until the instruction that is causing the hazard clears the pipeline. Instructions issued before the stalled instruction are also stalled, while instructions issued earlier than the stalled pipeline must continue or the hazard will never clear the pipeline. [17]

3.1.1 Structural Hazards

In a pipelined processor, the parallel execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some instruction combinations cannot be accommodated because of resource conflicts, the processor has a structural hazard. [17]

Structural hazards are most commonly avoided with hardware design techniques, by ensuring that there are sufficient functional units to run all possible combinations of instructions on the pipeline. Some of the more common causes of structural hazards arise when some part of the pipeline does not run at the same speed as the rest, for example a floating point arithmetic unit might take more than one clock cycle to complete at higher frequencies. Some pipelined processors may have a shared memory for data and instructions, this will most certainly cause bubbles in the pipeline since instructions should be fetched on every clock cycle and data memory accesses are quite common. [17]

When designing processors, but especially ASIPs, the decision whether to eliminate some structural hazard is not always so clear. Perhaps the design criteria call for lower power consumption, smaller area, or the stalls caused by the hazard are very rare. Clearly, as with any hardware design decision, building the right design always is a compromise of performance, power consumption and area.

3.1.2 Data Dependencies and Hazards

Determining how one instruction depends on another, is key in exploiting instruction-level parallelism. To effectively use the processors resources, we must determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls, assuming there are no structural hazards. If the two instructions are dependent, they cannot be executed in parallel and must be executed in order, although they may be partially overlapped. Determining whether an instruction is dependent on another is critical. There are three types of dependences: *data dependences*, *name dependences* and *control dependences*. [17]

Instruction A is data dependent on instruction B if:

1. Instruction B produces a result that may be used by instruction A .
2. Instruction A is data dependent on instruction C , and instruction C is data dependent on instruction B .

From the second condition, we see that data dependency chains are also possible and this chain can potentially lead through the whole program. [17] We saw an example of data dependency that would cause a data hazard earlier in chapter 3.1.

Dependencies are not inherently evil or something to be removed, but a property of programs. Whether a given dependence results in an actual hazard and stall, are results of the pipeline hardware and the software compiler. Data dependencies tell us three things: the possibility of a hazard, the order in which results must be calculated, and an upper bound on how much parallelism can possibly be exploited. Dependencies resulting in stalls can be eliminated with scheduling, which can be done by both the hardware and the compiler. [17]

The second type of dependence is a name dependence. These occur when two instructions use the same register or memory location but there is no flow of data between the instructions related to that location. Two types of name dependencies can be observed between instruction A that precedes instruction B in program order:

1. An anti-dependence between instruction A and instruction B occurs when instruction B writes to a register or memory location that instruction A reads.
2. An output dependence occurs when instruction A and instruction B write to the same register or memory location.

In both cases the ordering between the instructions must be preserved to maintain the dependence. In name dependencies, there is no value being transmitted between the instructions, so the instructions may execute in parallel or be reordered, if the register or memory location is changed so the instructions do not conflict. This renaming can be done on both hardware and by the compiler. [17]

Data hazards exist whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would cause the instructions to execute with non-deterministic behavior. Data hazards can be categorized to three classes. Again, consider two instructions A and B , with A preceding B in program order. Now the possible data hazards are

1. Read after write (RAW), B tries to read a source before A writes it, so B gets the old value.
2. Write after write (WAW), B tries to write an operand before it is written by A . The writes may end up in the wrong order, leaving the result of instruction A rather than instruction B in the destination. WAW hazards may only occur in pipelines that write in more than one pipe stage or allow instructions to proceed even when a previous instruction has stalled. These hazards are caused by output dependences.
3. Write after read (WAR), B tries to write to a destination before it is read by A , thus giving A incorrectly the value resulting from B . These hazards are caused by an anti-dependence.

Read after read is not a hazard, since no register or memory locations change values. [17]

3.1.3 Control Dependences and Hazards.

The last of the three dependencies mentioned in the previous section is a control dependence. Control dependencies determine the correct ordering of instructions, in respect to branch instructions so that instructions are executed in correct program order and only when they should be. All instructions are control dependent on some set of branches, or conditional jumps, and in general, control dependencies must be maintained to preserve program order. [17]

For example, in a C-esque code segment

```
if (A) {
    foo();
}
else if (B) {
    bar();
}
```

function *foo* is control dependent on statement A and function *bar* is control dependent on statement B but not on A .

Control dependencies impose two constraints for maintaining correct program order:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

By preserving strict program order, we ensure that control dependences are also preserved. However, we may be willing to execute instructions out of order and break one of the two constraints, if we can do so without affecting the correctness of the program. This makes program order non-critical to program correctness, instead *exception behavior* and *data flow* are the two properties that must be maintained at all cost. [17]

Maintaining exception behavior means that the reordering of instructions must not cause any new exceptions in the program. For example, in the following MIPS-style assembly code:

```

      ADD    R2, R3, R4
      BEQZ   R2, L1
      LW     R1, 0(R2)

L1:   ADD    R5, R5, R5

```

There is again a data dependence on register *R2*, if do not maintain this dependence the *BEQZ* (branch if equal to zero) instruction could incorrectly branch or not to *L1*. We might be tempted to move the *LW* (load word) instruction before the *BEQZ* instruction, but this might cause a memory protection exception. So, even though there is no data dependence preventing us from exchanging the *BEQZ* and *LW*, we must maintain the order due to the control dependence. [17]

The second property for maintaining program order is data flow. Data flow is the actual flow of data values among instructions that produce results and those that consume them. Branches make data flow dynamic, since they allow the source of data for a give instruction to come from many points of the program. Hence, program order determines which preceding instruction will deliver a data value to an instruction and program order is ensured by maintaining the control dependences. [17]

If we maintain strict program order, this would mean that for all branches we would stop issuing instructions to the pipeline until the branch has been evaluated. Otherwise, there could be a control hazard. Control hazards are prevented with a hardware control hazard detection unit, that would allow you to keep issuing instructions as if the branch was or was not taken, and if your guess was incorrect flush the pipeline of the wrong branch's instructions before they do any harm. There are hardware and software techniques to improve the accuracy of branch prediction, based on previous iterations of the branch.

3.1.4 Dynamic vs. Static Scheduling

To fully utilize a pipelined processor, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To

avoid stalls, execution of dependent instructions must be separated by a distance in clock cycles equal to the pipeline latency for the first dependent instruction. Finding and reordering unrelated instructions to issue, while maintaining exception behavior and data flow, is called scheduling. Scheduling can be done statically at compile time, or dynamically at run time. Static scheduling is naturally a compiler technique, while dynamic scheduling relies on hardware to issue instructions and avoid hazards at run time. Processors using the dynamic approach, including the Intel Core series, dominate in the desktop and server markets. Static scheduling is mostly used in fields where energy efficiency is key, like the personal mobile device market. [17]

One compiler technique for exposing ILP that both static and dynamic scheduling can employ, is loop unrolling. Say we have a C-code loop, which adds a scalar to a vector:

```
for (i = 999; i >= 0; i--)
    x[i] = x[i] + s;
```

We can see that the loop is parallel since there are no dependencies between the elements of vector x . In most pipelines, loop structures create a few clock cycles of overhead per loop iteration. This is a result of incrementing the loop variable and conditional branching. As a result, there is a data dependency which causes bubbles in the pipeline. To reduce the performance hit from loop overhead, in situations where there are no data dependencies within the loop body, we can replicate the loop body multiple times and adjusting the loop termination condition. If the upper bound of the loop is known we can simply eliminate the loop completely by fully unrolling the loop, in our previous example this would mean replicating the loop body a 1000 times. Since in real programs the upper bound of the loop is not always known at compile time, fully unrolling loops is not always possible. Suppose we have a loop upper bound n , and we would like to unroll the loop to make k copies of the body:

```
for (i = 0; i < n mod k; i++)
    x[i] = x[i] + s;
for (i = n mod k; i < n/k; i += k) {
    x[i] = x[i] + s;
    x[i+1] = x[i+1] + s;
    .
    .
    x[i+k] = x[i+k] + s;
}
```

The first loop has the original loop body and executes $n \bmod k$ times, while the second loop is unrolled by a factor of k and iterates n/k times. This is similar to a technique called strip mining, and is good for larger values of n since most of the execution time would be spent in the unrolled loop. [17]

A statically scheduled pipeline fetches an instruction and issues it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction. If

there is a data dependence the pipeline could be modified to bypass or forward the dependent data to an earlier stage in the pipeline before the instruction completes. If this is not possible the pipeline control logic will stall the pipeline and no new instructions are fetched or issued until the dependence is cleared. [17]

In dynamic scheduling, the hardware will try to rearrange the instruction execution to reduce stalls while maintaining data flow and exception behavior. There are several advantages to dynamic scheduling. It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline, which is of significance since most software is distributed by third parties in binary form. Also, it enables handling some dependencies that might be unknown at compile time, such as: memory-references or dynamic linking. And most importantly, it allows the processor to tolerate unpredictable delays, like cache misses, by executing other code while waiting for the miss to resolve. While dynamic scheduling offers many advantages, these gains are at the cost of a significant increase in hardware complexity. Compiler pipeline scheduling, as in statically scheduled pipelines, can also be utilized with dynamically scheduled pipelines. [17]

3.2 Multiple-Issue Architectures

With a pipeline using the techniques to avoid stalls described in the previous section, the performance would be capped when we reach a stage where we can issue one instruction every clock cycle. To climb this performance wall, we need to issue more than one instruction per clock cycle. These processors are called multiple-issue processors, and they can be categorized into two distinct classes, based on the scheduling method employed:

1. Statically scheduled multi-issue processors
2. Dynamically scheduled multi-issue processors

Figure 6 shows a comparison of the different multiple-issue architectures. Superscalar processors issue a varying number of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled. Very long instruction word (VLIW) processors, in contrast, issue a fixed number of instructions as one large instruction. VLIW processors are statically scheduled by the compiler. EPIC refers to explicitly parallel instruction computer and is a similar architecture as VLIW. Where VLIW has an exposed pipeline, EPIC pipeline is protected meaning if you try to use an instruction's results before they are ready the pipeline will stall, so the scheduling method can be considered as semi-static. [17]

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM POWER7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 6. Multiple-issue processor comparison [17].

Superscalar architectures have been very successful in exploiting parallelism in general-purpose processors. VLIW architectures have been effective in domains like signal processing, where compilers are able to extract enough parallelism. VLIW architectures usually offer lower power consumption. [13]

3.2.1 Very Long Instruction Word Architecture

VLIWs use multiple, independent functional units. To issue commands to these units, a VLIW packages multiple independent instructions, to one very large instruction. To keep the functional units busy there needs to be enough parallelism in a code segment to fill the available operation slots. This parallelism can be uncovered by unrolling loops and scheduling code within a single larger loop body [17]. Figure 7 shows an example VLIW instruction word for a VLIW where there are 4 functional units $FU0 - FU3$. This is a very simplified example where each instruction has the same width.

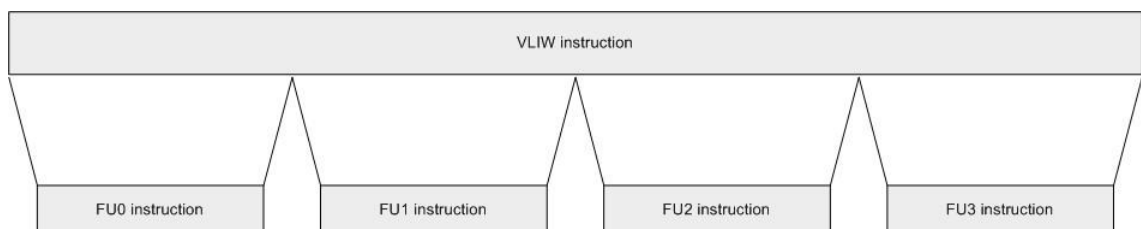


Figure 7. An example VLIW instruction word.

There are generally thought to be two static scheduling methods for compilers to use: local- and global scheduling. Local scheduling relies on finding parallelism within a code block, while global scheduling tries to exploit parallelism even across branches. Global scheduling is clearly much more complex, since moving code across branches is expensive. [17]

VLIWs are not without their issues, technical problems arise from increase in code size and the limitations of lockstep operation. Code size is increased due to two factors: unrolling loops multiply the original loop body by the unrolling factor, and if there are no instructions to issue to each functional unit they still need to be issued bubble or No Operation (NOP) instructions. This translates to wasted bits in the instruction word. Code size increase can be combated with instruction word encoding. Instructions can be compressed in main memory and expanded when they are decoded by the processor. Instructions could also share an immediate field across all functional units, which can reduce the overall instruction word width [17]. In VLIWs, there is a lot of room, and need, for clever optimizations since every bit saved translates to smaller instruction memory.

VLIW lockstep operation can lead to unnecessary stalls, if there is no hazard-detection hardware. In lockstep, all functional units move from one pipeline stage to the next at the same time, and if there is a stall in one unit, all other units must stall as well. This can lead to very poorly performing code, especially if multiple units can use the same memory or register at the same time. VLIW functional units can also be made to function in a more independent manner, where units could work asynchronously after an instruction has been issued. This of course requires a hardware hazard-detection unit, in addition to static scheduling methods. [17]

VLIW architectures are highly modular, and such are a great candidate for ASIPs. Functional units can be added and removed to suit the target application. One common way to utilize VLIW in ASIPs is to have one dedicated scalar lane for your general-purpose computing, and adding functional units and instructions to accommodate more computationally intensive operations, like floating point and vector arithmetic.

3.2.2 Transport Triggered Architecture

Transport Triggered Architectures (TTA) can be thought of as a special case of VLIW architecture. As in VLIWs, a single instruction word is issued each clock cycle, but it contains multiple independent parallel instructions. However, TTA instructions do not issue RISC type operations in the instruction word, instead each instruction specifies a data move and the actual operation is executed as a side-effect of the data move [18]. Corporaal et al. [18], introduced TTA in 1998 and targeted TTA specifically for ASIP use.

TTAs can be viewed as a set of functional units and register files, connected by an interconnect network. As with VLIWs, TTAs are statically scheduled multiple-issue processors. Figure 8 shows a generic TTA processor structure, functional units and register files are connected to the interconnect with one or more input and output sockets, there is also the network controller (NCTL) which controls the pipelining of the network.

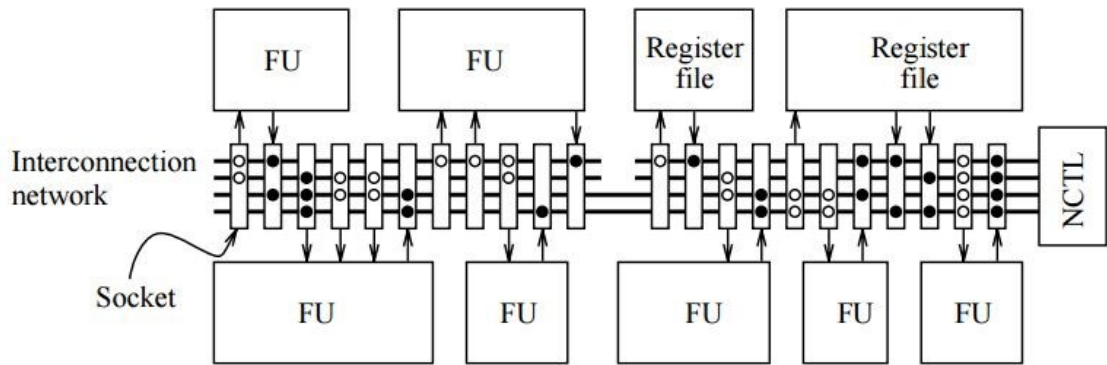


Figure 8. General structure of a TTA processor [18].

As mentioned earlier, TTA instructions issue data moves from source to destination. Thus, the instruction word consists of one data move operation for each bus. For each bus there are only three fields in the instruction: *src*-field indicates from where the move happens, *dst*-field indicates where and the *i*-field indicates whether the source is an immediate or a register [18].

TTAs take static scheduling to its very limit. The whole architecture can be seen from Figure 8, there are no hardware hazard detection units or other dynamic scheduling constructs. This makes TTAs, in theory, very light weight in terms of hardware complexity and power consumption. Performance depends solely on the functional units selected, and the compilers ability to expose ILP in the code to efficiently run it on the selected TTA processor.

3.3 Data-Level Parallelism

Single instruction, multiple data (SIMD) architectures usefulness depends entirely on the level of data-level parallelism inherent in an application. SIMD has been prolific in image and sound processing, as well as matrix-oriented computations. Since, parallel programming can be quite confusing for even the experienced programmer, SIMD has the added benefit over multiple instructions, multiple data (MIMD) architectures, that the programmer can continue thinking sequentially while achieving parallel speedup with parallel

data operations. SIMD architectures come in three variations: vector processors, processors with multimedia SIMD instruction set extensions and graphics processing units (GPUs). [17]

3.3.1 Vector Processors

In vector architectures instructions operate on vectors of data, as opposed to singular values in scalar architectures. Vector instructions can deliver high performance without the design complexity, or high power consumption, of out-of-order superscalar processors [17]. While vector architectures may not be complex, the added performance does not come without cost. Either vector loads and stores must be heavily pipelined, and slow, or the data busses need to be widened to accommodate vectors. This leads to more flip-flops and wires, and can make the design quite congested. To meet design timing requirements, increases in area are to be expected.

A major advantage of vector architectures is that it allows software to pass a large amount of parallel vectorizable work to hardware, using only a single instruction [17]. A basic vector architecture will have only one lane for vector operations, which implies that the functional unit will operate on the vectors one element at a time. To increase performance, the functional unit can be replicated to create multiple lanes, which can all operate at the same time. Vector elements are not dependent on other elements in the vector, so vector architectures do not need to perform costly dependency checks, as superscalar processors require [17]. Vector architectures allow the designer to freely optimize between area, power and clock rate, while meeting performance requirements. For example, increasing area by doubling the amount of vector functional units, and halving the clock frequency can save you in terms of power consumption, while retaining the same performance.

Another advantage for vector architectures is that compilers can tell programmers at compile time whether a section of code will vectorize or not. Most compilers will allow you to add instructions for the compiler to your code, which can help the compiler find the right balance in unrolling loops or letting the compiler assume independence between operations. Success in vectorizing code, is entirely dependent on the program structure: are there data dependencies within the loop, or could they be restructured so that the dependencies disappear. [17]

When comparing vector architectures to multi-issue architectures, there is no clear advantage for either. Vector architectures are faster with vectorizable code than multi-issue, but vector architectures struggle with control oriented code. One popular alternative is to extend multiple-issue architectures with vector operation lanes.

3.4 ASIP Tools and Methodologies

As we noticed in the previous chapters, processor architecture plays a significant role in determining our systems performance. After we have made our architectural decisions, we still need to describe our processor in a formalism, from which we can extract a netlist with synthesis for place and route. To do this with an RTL hardware description language, like VHDL or Verilog, is quite the effort, and keeping design exploration iterations quick nigh impossible. To tackle this issue, there are machine description languages that will allow us to describe our processor on a higher abstraction level, which should speedup design exploration. Two of the more prominent machine description languages are discussed here: LISA and nML.

3.4.1 LISA

LISA was introduced by Zivojnovic et al. [19] in 1996, to cover the gap between standard instruction set architecture (ISA) models, used in compilers and instruction-level simulator, and detailed description languages such as VHDL. LISA is an operation-level description of the pipeline, which is able to model even more complex interlocking and bypassing techniques. Instructions in LISA are made of multiple operations which are defined as register transfers during a single control step. LISA models are bit-accurate, and the resulting timed model delivers instruction-, clock, or phase-accurate timing for simulation purposes.

LISA development was headed by Aachen University in Germany [20]. Synopsys had a tool suite based on LISA 2.0 language, but they no longer seem to support or sell that product.

3.4.2 nML

nML is based on a mix of behavioral and structural paradigms, which results in a formalism that is akin to the abstraction level of a programmer's manual. The formalism declares all storage entities, which results in a skeleton of the processors structure. nML also declares all register transfers between these storage elements, describing the exact execution behavior of the processor. nML allows for bit-true modelling of the target processor, control over program flow and instruction encoding. [21]

In nML, a hierarchical structure can be imposed onto the instruction set description and a top-down approach to instruction set design is advocated. The instruction set description is partitioned to two types of rules:

1. *OR-rules*, which list all alternatives for an instruction part
2. *AND-rules*, which describe the composition of an instruction

These rules form a tree structure, as shown in Figure 9, here OR-rules are shown with balloons and show all the available instructions, while the AND-rules show the parameters in each instruction.

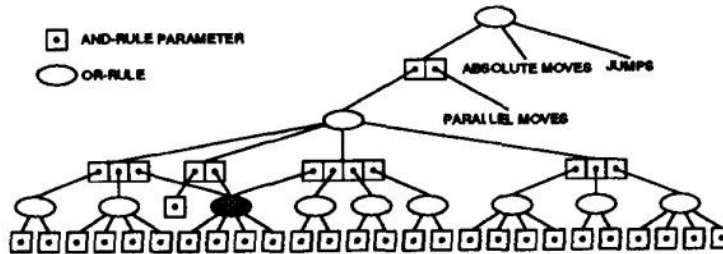


Figure 9 An nML instruction set tree [21].

The AND-rules define the semantics of the instructions with three attributes: action-attributes specify the execution behavior, image-attributes the binary encoding and syntax-attributes describe the assembly language mnemonics. nML language was introduced by Fauth et al. in 1996. [21]

3.4.3 ASIP Designer by Synopsys

ASIP Designer is a tool suite by Synopsys that uses the nML machine description language to model ASIP instruction-set architectures in a clock cycle- and bit-accurate way. The tool suite offers a wide range of features, including: automatic software development kit (SDK) generation, instruction-set simulator, for both cycle- and instruction accurate simulation, on-chip debug hardware, instruction-set architecture profiling and automatic generation of synthesizable VHDL and Verilog. [16]

ASIP Designer design flow is depicted in Figure 10, here we see that the tool suite only requires user input for only two parts: nML processor architecture and C/C++ code for the actual application. The steps 1 and 2, will make up most of the design effort, going back and forth to optimize the processor architecture to meet performance requirements set for our system. While the figure does not show it, going back to refine the processor

model is more than likely in a real ASIC project after steps 3 and 4, to meet area and power requirements.

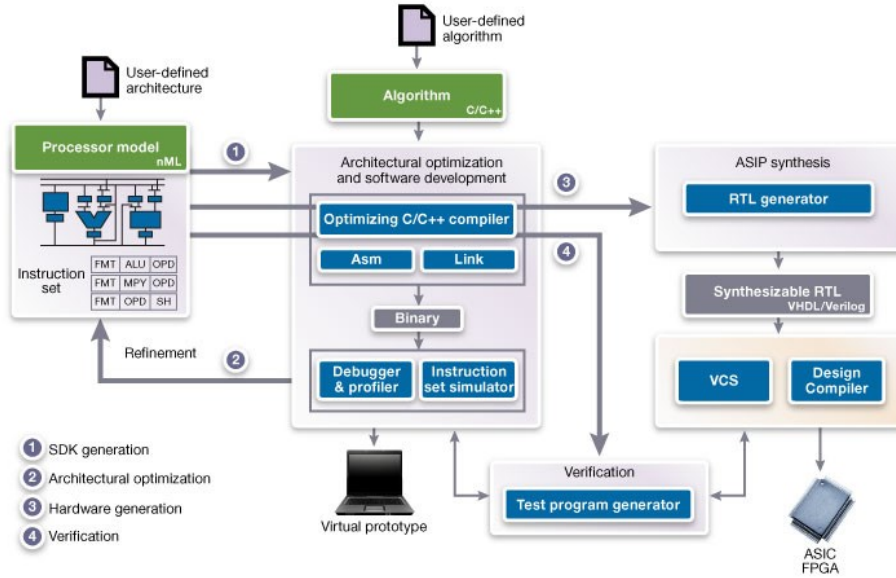


Figure 10. Synopsys' ASIP Designer design flow [16].

ASIP Designer also ships with multiple example ASIP designs in nML source code format, that can be utilized for fast initial design exploration. Also, a RTL verification environment can be generated from our C/C++ test application where the same binary and test data is loaded to the memories of the targeted ASIP. This allows us to perform verification also at the RTL abstraction level. [16]

3.4.4 TTA-Based Co-Design Environment

TTA-Based Co-Design Environment (TCE) is a toolset for designing and programming customized processors based on TTA. TCE is developed by the Customized Parallel Computing group at the Department of Pervasive Computing of Tampere University of Technology (TUT). TCE features a LLVM [22] based compiler for TTA architectures, an instruction-set simulator with cycle count accuracy, processor and program image generation and support for automated, semi-automatic or manual algorithm implementations. This package is wrapped together into an Integrated Development Environment (IDE), for graphical user interface (GUI) based editing of the TTA structure, namely register files, functional units and the interconnect between them. [15]

TCE is an open-source project, that is headed by the Customized Parallel Computing group at TUT, while this means that the toolset is accessible, it might also be lacking in

support. This could mean slower bug fixes and longer support request response times. These risks might affect the design's overall design quality, either directly with missed bugs in design or compiler, or indirectly in the form of pushed tape-out, due to some lack of support.

4. IMPLEMENTATION

Now that we have a firm grasp on the application area, and desired implementation technology, let's reiterate the design requirements. Our target is to design a hardware solution for cognitive radio (CR) ASICs, to perform LAA CCA decision making from captured data on the selected carrier frequency. In addition to performing energy detection and the LAA LBT algorithm, our design should meet three key requirements:

- *Performance*, LAA CCA is a contention based channel access algorithm. To make good decisions, our chip needs to know as soon as possible if the carrier is free. This will also improve interoperability, since reduced the latency from the time our carrier contention algorithm deems the carrier idle until transmission should decrease the chance of collisions.
- *Flexibility*, due to risks of changing 3GPP specifications from now until 5G is fully deployed, we need our solution to be amenable towards these changes. This will allow us to compete for better market position, if our chips are first solutions on the market that fulfill the specifications. To accommodate this, our chips RTL freeze would need to be well in advance of the potential unveiling of 5G in Tokyo Olympics, to allow the product to be developed around it.
- *Low power consumption*, while advantageous in most designs, lower power consumption IPs in huge ASICs, which CR chips tend to be, is a key feature. While one might be able to fit all of the logic required on an ASIC, keeping it cooled to a temperature where it will still function deterministically, can be a larger challenge if the designers do not keep power consumption in mind.

With these design criteria in place, let's look at what exactly we need to do to meet these requirements. First, we must determine what our performance requirement means in terms of processing complexity and what our timing budget to meet it is. Then, we explore some options we have to meet the performance requirement, while keeping flexibility and low power consumption in mind.

4.1 Processing Steps

To perform LAA CCA described in chapter 2.1, we must maintain a LBT state machine for each carrier on which we would like potentially transmit. Then to advance the state machine, we must do energy detection for the defined defer durations. For this we need to capture data on the physical transmission medium, in our case radio waves, and perform energy detection as defined by algorithm (12). Since our radio chip captures data in time domain, we need to transform the captured samples to frequency domain, to utilize algorithm (12). For this, we must compute the Discrete Fourier Transform of the sample

sequence, or FFT to reduce time complexity of the algorithm. The FFT could be calculated with the decimation-in-time algorithm we defined in equations (9), (10) and (11).

Since, LTE and WLAN transmission timings are not synchronized, we cannot know the exact edges of WLAN 9 μs slots. To get around this problem, we can take multiple consecutive smaller time slots and assess the channel state as a combination of these results. Figure 11 shows us this concept with measurement slots of $\sim 4,2 \mu\text{s}$, CCA results are shown as B for busy, I for idle and ? for results that might be inconclusive.

WLAN slots	Busy		Busy		Idle		Idle		Busy		Busy		Idle	
Sampling slots	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs	4,2 μs
CCA result	B	B	B	?	I	I	I	?	B	B	B	I	I	I

Figure 11. CCA concept.

To try and synchronize our timing to WLAN, we might consider using even smaller measurement slots to detect the edges when WLAN transmission starts. In this thesis, we will focus on the above concept where our measurement slots are around $4,2 \mu\text{s}$ and there are 1024 samples within that slot, which leads to a sampling rate of $\sim 244 \text{ M samples/s}$. Also, we will limit our multi-carrier LBT approach to type A, described in chapter 2.1, which is the more processing intensive alternative, where all 8 carriers perform the full CCA.

With this concept, our processing steps are as follows:

1. Perform 1024 sample FFT.
2. Calculate signal power, for all carriers, and compare with energy detection threshold value to determine carrier state.
3. Advance CCA state machine, if enough consecutive idle slots have been detected.

All three steps need to be processed within half of the WLAN slot, $4,5 \mu\text{s}$, this results in 6,6% of samples not being processed. If we leave a $0,5 \mu\text{s}$ time budget for direct memory access (DMA), to move samples from the chips capture memory to our sampling memory, that leaves us with a processing time budget of $4 \mu\text{s}$.

4.2 Implementation Approaches

To implement meet our performance and low power criteria, one might be tempted to start implementing our LAA CCA IP, with custom RTL using VHDL or Verilog, but since these approaches are non-flexible, we can rule them out. While ASIPs, perhaps offer lower performance and higher power consumption than custom RTL, the tremendous increase in flexibility, through programmability, should outweigh the downsides for our application area. Since, ASIPs can be designed to fit any mold, one could also consider a hybrid implementation approach, where an ASIP is supplemented with custom IP blocks for processing part of the algorithm.

4.2.1 ASIP with FFT Accelerator

Hybrid approaches sacrifice flexibility, by separating a part of the algorithm to be run on a specialized IP, for increased performance and probably lower power consumption. One way to design a hybrid ASIP would be to incorporate a co-processor that would run part of the application asynchronously from the ASIP operation. But since our application is highly sequential in nature, it would make more sense for us to break off a part of that sequence to be run as a separate IP.

One good candidate would be the FFT, since transforming samples from time domain to frequency domain is something we must do regardless. This would leave energy detection and the control oriented CCA state machine for the ASIP. Figure 12 shows a block diagram of the processing chain for this approach, here the time domain samples are fed through the FFT accelerator, which stores the resulting frequency domain samples in the ASIPs sample memory, then the ASIP calculates the signal power for all the 8 carriers and produces the CCA results. The FFT could be run in parallel to the ASIP, if the sample memory is doubled in size so that the FFT would on first iteration write from sample location 0-1023, and the ASIP would start processing those, while on the second iteration the FFT would write from sample location 1024-2047, and while the ASIP processes those the FFT starts from the beginning again.

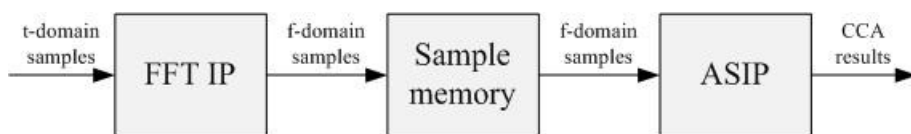


Figure 12. Simplified block diagram with an FFT IP and an ASIP for LAA operation.

The FFT is a good candidate for separation from rest of the chain, since there is little variation in the algorithm, and only limited control needed: configure FFT size and feed

samples through. The other candidate would be energy detection, but there can be more variation on where the relevant carrier frequencies reside on the channel. This might mean that the ASIP or perhaps the chips main processor would need to configure it quite often.

4.2.2 ASIP-Only Solution

The last option after fully custom RTL design and ASIP coupled with an FFT IP, is to do the full algorithm processing on a single ASIP. Figure 13 shows a block diagram of the processing chain, while this time the DMA unit would write the time domain samples directly to the ASIPs sample memory, from where the ASIP would fetch them and start processing them: first performing FFT on them, then energy detection followed by the CCA state machine and reporting results as on output.

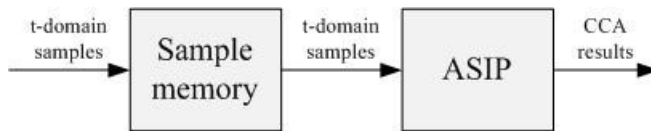


Figure 13. *Simplified block diagram for a LAA ASIP.*

For the ASIP-only implementation we would like to double the maximum supported measurement window in samples. In our initial case, this would be $2 * 1024$ samples, but if we would like to support also bigger measurement windows this could also be $2 * 2048$.

4.3 ASIP Implementation

While the hybrid approach does at first glance seem to offer improved performance due to the parallel FFT and ASIP, this might not necessary be as beneficial as it seems. The fact of the matter is that there are only 1024 samples coming in every $4,5 \mu s$, so if the FFT and ASIP would run in parallel this would mean that we either are missing our performance requirement of processing all three steps for 1k samples, or we are adding latency to our calculation, while not increasing throughput. In the latter case this would mean that the FFT would process 1k samples every $4,5 \mu s$, and the ASIP would process those samples in the next $4,5 \mu s$ slot while the FFT starts to process the next 1k sample batch at the same time.

Due to this, and a desire to emphasize flexibility through having the entirety of the application be re-programmable after fabrication, it was decided to focus on ASIP only approaches for the design. For the processor architecture, a few decisions were easy to make. First, since the ASIP will only be running code for LAA CCA the software is closer to

firmware and should be statically scheduled at compile time to reduce hardware complexity. For the same reason, the instruction memory does not need to be updated dynamically and can be filled once with the code binary at system start up. Second, to utilize ILP the processor needs to be pipelined and third, due to the fact that FFT and energy detection algorithms are easily vectorizable, the processor should either be a vector processor or have a specialized vector lane.

For the tool suite ASIP Designer from Synopsys was chosen, since it is not limited to just one architecture paradigm, like TCE, but can be used to design any type of processor, with fast iterations between architecture changes and SDK generation. Also, since TCE is an open-source tool suite there is a risk that we might receive limited support for the tools, while for ASIP Designer there are a number of experienced users working at Nokia, and Synopsys provides support quite quickly.

4.3.1 Design Exploration

To start design exploration, we deemed the FFT to be the biggest performance bottleneck of the application, since energy detection is only squaring and accumulating the signal power, thus time complexity for FFT and energy detection are $O(n \log n)$ and $O(n)$ respectively. To start, we needed a processor architecture that could perform 1k sample FFT, with time to spare for energy detection and CCA state machine advance, in 4 μ s with our target clock frequency of around 600MHz.

Initially we started looking at the example cores provided by ASIP Designer, to try and look for a good starting point. Multiple cores were tried: some specialized for low power FFT operation with limited control operations, more general purpose architecture with separate lanes for scalar and vector operations, and a VLIW processor. As one might expect from example cores aimed as starting points for design exploration, none of the architectures could provide the performance we needed. Others would have required only some work to meet performance, but were lacking the scalar portion of the processor that we require, while others needed to be stripped down of unnecessary features like every imaginable bitwise operation and still would have needed performance increases of around 70%.

The most promising among these candidates was the architecture with separate scalar and vector lanes, which needed a sizable performance increase. The vector width could be expanded quite easily with little effort and parallelism within the vector lane could be increased by widening the instruction word and issuing more instructions per clock cycle, as in a VLIW.

The final contender was from a Nokia internal design, that was being developed at Nokia Bell Labs. This processor was aimed for high performance FFT, of any size and multiple different radix were supported. This architecture was a highly parallel VLIW processor

with specialized and vectorized lanes for reordering, twiddle coefficient operations and vector butterflies, and it included a small scalar lane.

The processor from Nokia Bell Labs, while promising great performance, was initially ruled out since the highly parallel VLIW was thought to be inefficient for other application areas than FFT. This would lead to a lot of no operation (NOP) codes in the instruction memory, so a lot of wasted memory and processing power. There were some concern as to the area of the processor. But after support from the cores designer at Nokia Bell Labs, we received a version of the core that had halved the vector width from 16 complex samples to 8, which still easily offered performance within our timing budget, and significantly lowered the area of the core. We also noted that, while the instruction memory still holds a lot of NOP-codes within one instruction, the number of instructions needed for our application was surprisingly low, which meant that while the instruction memory required was very wide, it was not very deep.

Result of the initial design exploration was to take the core developed at Nokia Bell Labs, and develop it further to meet our LAA application requirements. Some of the additions that needed to be made were: instruction for vector squaring, since the unit already had vector multiplier for the FFT application, only a multiplexer and wires from one register to both multiplier inputs were needed, also element-wise loads from vector registers to scalar ones was needed to extract the accumulated signal power for energy detection. A general-purpose IO interface was also added, for controlling DMA and signaling upstream the results of CCA.

4.3.2 Architecture

Finally, the resulting ASIP is a pipelined, VLIW processor with vector operation lanes specialized for FFT operation, and including instructions enabling our full application to be run on the core. The scalar lane of the core is sufficiently equipped to handle most control structures found in C, and useful C-libraries, such as string, rand and sort, compile without problems.

Since the ASIP is Nokia proprietary information, that might be included in future Nokia ASICs, we cannot explore the detailed architecture or instruction set here. But to compare the solution to our three key criteria from the beginning of this chapter:

1. *Performance:* 1024 sample FFT and energy detection for all 8 carriers can be run in $\sim 2,5 \mu s$, and the CCA state machine only executes 7 instructions on the average iteration, which puts us way under our time budget.
2. *Flexibility:* With our ASIP only approach the entire application is software controllable and should be able to accommodate any changes in the LAA specification.

3. *Low power consumption:* Since our design has quite a bit of room in the timing budget, it might be worth looking into lowering the clock frequency or at the very least providing this option. This could be a software controllable clock divider outside of the ASIP, that drives the ASIP clock and reset.

5. VERIFICATION AND ANALYSIS

In this section, we will go through the verification methods that were used for our ASIP we designed in the previous chapter, and try to quantify the results with synthesis and performance metrics. Since the core is still under development, these results are not final but since we have already met our performance target it is quite certain that the area, power and performance should not increase after this point. Mostly, further optimization work for the core is ongoing, at the time of writing this thesis.

5.1 Verification

Verification of ASIPs is somewhat different than software testing, or hardware verification. Since, what we are developing is a processor that runs our software, the entire design and debug process involves hardware-software co-simulation. For this section, we will skip software only unit tests, and focus on verifying the HW/SW combination as a whole. For ASIPs two levels of verification were employed: cycle-accurate instruction set simulation and RTL simulation. For both we use the same test application, since our software is closer to firmware, or a kernel module, it will run indefinitely after the ASIP has been configured.

Program 1 shows the structure of the indefinite loop that will execute our application. First, we must wait for the *samples_ready* interrupt that tells us the DMA has finished loading the captured time-domain samples to the sample memory, then we should clear the interrupt to preserve correct execution on the next iteration. Second, we transform the samples from time-domain to frequency-domain with FFT, in program 1 denoted with *run_1k_fft()*. Third, we must calculate each carriers signal power for that measurement window with function *power(int start_bin, int stop_bin)*, the function parameters indicate over which samples, or frequencies, should the power be calculated for that carrier. This is due to the fact that WLAN channels have a guard band at the end and beginning of the frequency band. Finally, we complete energy detection for each carrier with function *cca(int power, CCA_state state)* by comparing the obtained signal power value to the energy detection threshold, and radar detection threshold as well for DRS. Passed parameters for *cca()* are the calculated signal power and the carriers state machine variable.

```

1  while (1){
2      if (samples_ready == true){
3          //Clear the samples_ready interrupt
4          clear_interrupt();
5
6          //Perform FFT for 1024 samples:
7          run_1k_fft();
8
9          for (int i = 0; i < num_of_carriers; i++){
10             //Calculate the signal power for each carrier:
11             carrier[i].power += power(carrier[i].start, carrier[i].end);
12
13         }
14
15         for (int i = 0; i < num_of_carriers; i++){
16             //Compare signal power to energy detection
17             //threshold and advance state machine
18             //if necessary:
19             cca(carrier[i].power, carrier[i].state);
20
21         }
22     }

```

Program 1. *C++ representation of our test program structure.*

Since the test application is quite simple but the LAA CCA can be quite complex with multiple carriers contending for the transmission medium, our test data will determine our design's completeness. This means generating very large sets of test vectors, to test all the different possible state transitions it would take very long to generate such test data. In this thesis, we will only verify our system with a few test vectors, of known idle or busy states, that we loop through the memory.

Initial hardware/software co-simulation was done with a cycle-accurate instruction-set simulator, which ASIP Designer generates from our nML model of the processor. The simulator has the feature set you would see in most other simulators: step-by-step execution, breakpoints, memory, register, and variable states that can be modified at run time for quick testing. Also, the simulator shows for each pipeline stage the instruction that it is running for the current clock cycle. ASIP Designer also features an instruction-accurate instruction set simulator, but to get the correct performance figure in clock cycles the cycle-accurate simulator was used.

After the instruction-set simulator we moved on to RTL simulation, to see that the instruction-set simulators behavior matches the generated RTLs. For RTL simulation, ASIP Designer offers the possibility to generate the simulation environment to match the test bench of your instruction-set simulator. This means that all the same data sets are loaded to the generated RTLs memories in the simulator. We did not see mismatches with the RTL simulation and instruction-set simulation, for RTL simulation we used VCS [23] from Synopsys.

Linting and formal verification were not incorporated in this design's verification. Linting was ignored since the RTL is generated by ASIP Designer tool, which is not meant to be modified at RTL level, or read by a human for that matter, thus linting would offer no benefit. With formal you might be able to verify without a doubt that some hazards cannot physically happen, or in our case that they would happen, but our processor is statically scheduled so it is our compilers responsibility to avoid hazards in execution.

5.2 Synthesis

To generate RTL for synthesis, we used ASIP Designers Go tool, which does the nML model to synthesizable RTL translation. Go can generate both VHDL and Verilog hardware description languages. By default, Go uses VHDL-93 and Verilog-1995 standards, but options to use other versions are provided. For Go, there are a number of configuration options to give the tool directives for RTL generation, these options span from simple naming convention rules to more complex optimization options for reducing design critical path and low power options.

Go provides the option to generate a basic scripting environment to run synthesis, these synthesis scripts are not meant as a final synthesis environment. For synthesis, we used Design Compiler [24] from Synopsys, and our own scripting environment that maps the resulting netlist to our target technology library. Suffice to say, our technology library is quite newer than the example library that the Go synthesis scripts use.

While the area, performance and power figures from synthesis are proprietary information, and the technology library's non-disclosure agreement prohibits us to share the exact figures, we can compare the results to other designs with the same technology library to quantify our results.

In Table 2, we compare three different designs in term of area, power and performance. The first solution is based the previously described ASIP with FFT accelerator solution, the other two solutions represent the ASIP only design solution, that we are developing, one with the original vector width of 16-complex samples and the other with the lower 8-complex sample vectors. All designs were synthesized with the same parameters, clock frequency, optimization level and technology library. The ASIP with FFT accelerator was chosen as the baseline for area and power consumption, comparing this ASIPs performance to the other two would offer no value since it is designed to work with a different timing concept, thus we just compare the performance of the other two ASIP only solutions. For each column in Table 2, the value 100% identifies that solution as the baseline for the metric. The ASIP-only solution with 8 sample vectors is compared to the ASIP with FFT accelerator solution, and the ASIP-only solution with 16 sample vectors is compared to the one with the smaller vector size of 8.

Table 2. *Relative design area, power and performance, of three ASIP solutions targeting LAA feature set.*

ASIP solution	Area	Power	Performance
ASIP with FFT Accelerator	100%	100%	N/A
ASIP-only, 8 sample vectors	+ 15,2%	+ 18,6%	100%
ASIP-only, 16 sample vectors	+ 60,5%	+ 62,2%	+ 42,2%

From the synthesis results, we notice that even though the ASIP with FFT accelerator solution is the leanest as we anticipated, adding the necessary parallel vector operations to the ASIP did not increase area significantly after we scaled the 16 samples wide vector operations, registers and memories down to 8 sample vectors. For our application, the performance of the 8-sample vector ASIP seems to hit a sweet spot area to performance ratio, since increasing the vector width to 16 samples only provides us with 42,2% more performance, while the area increases 60,5%.

While the power consumption is reported here, the results should be taken with a grain of salt, since this is just the power result from the synthesis tool and not from an actual power measurement tool. When we look at power consumption of the two ASIP-only solutions, we see that they scale quite linearly with the area increase, 62,2% increase in power consumption vs 60,5% in area.

To summarize, the ASIP-only solution with smaller vector width of 8-complex samples offers a great performance to area ratio, while maintaining competitive area and power consumption to the ASIP solution with an FFT accelerator replacing part of the processing. These results seem to confirm our chosen approach after design exploration as a good candidate for further development.

5.3 Further Development

For further development of the ASIP, there are some good features with our ASIP Designer tool suite that could be employed, to identify some areas that could be optimized. The tool set has a nice set of profiling tools, that could be used to find instructions that are not used much with our application. If these instructions are working in parallel with everything else, trials should be made to see what the performance hit would be, if their functional units would be integrated as part of some other functional unit. This would make the particular lane more serial, but if the performance hit is small and area gain non-zero, this ought to be considered.

The instruction-set encoding should be further optimized, and there might be possibilities to remove some bits from the instruction word in this manner. ASIP Designer includes a GUI for instruction set viewing, in a hierarchical fashion, which can be utilized here and some command line options for further analysis. Reducing the instruction word width

should have a significant impact on the area and therefore cost, since memories tend to be quite costly.

The verification environment would also need further development, while the auto-generated RTL simulation environment is a welcome addition, it is not sufficient for modern RTL verification. We need to build a universal verification methodology (UVM) environment, that will enhance our verification capabilities. UVM is an object-oriented verification environment based on SystemVerilog language, which is a higher abstraction level language than VHDL and Verilog, while still maintaining some their fine granularity [25]. With UVM we can define sequences to be run on the core, and the sequence types can be fully randomized or directed with constraint random.

6. CONCLUSIONS

In this thesis, we explored the 3GPP LAA DL feature, and identified the key processing stages and requirements for a hardware solution. We concluded that the hardware solution needs to perform FFT, energy detection, and maintain and update the CCA state machine for all 8 carriers. Additionally, we noted that, due to the fluid state of the 3GPP specification, we would like to provide a flexible solution that would be poised to support changing requirements.

We presented two approaches for the design, one pure ASIP solution, and one hybrid ASIP with a FFT accelerator attached. With further study of the two approaches, we deduced that the parallel nature of the hybrid ASIP, does not provide us added performance, unless we add one sample latency to the processing pipe. This approach did not promise improvements in throughput, so the ASIP only solution was chosen, since it was also more flexible.

In ASIP architecture exploration, we noticed that, a pipelined, multi-issue architecture with vector processing units could be a great candidate for our ASIP solution, and decided to choose a Nokia internal design for further development. The chosen core was a VLIW processor, which specialized in FFT operation, using vector operations, that we modified to accommodate energy detection functionality.

After verifying the functionality of the core on two levels, cycle-accurate instruction-set simulation and RTL simulation, we synthesized the proposed solution with two different vector widths. Comparison of the two cores to a previous solution, showed that the core with the smaller vector width offered great performance to area ratio, while still offering performance that had room in the time budget. With more optimization and verification effort, the ASIP solution introduced here, offers a great solution for the 3GPP LAA feature, that should adapt nicely to the upcoming specification, while providing great performance for its size.

All in all, the design exploration portion of this study proceeded quite nicely, but learning to use the ASIP Designer tools took some time, and learning nML significantly more. Documenting the requirements and theory already in the starting phase of this study, would have eased, not only the writing process, but the design effort as well, since during processor architecture evaluation, we went in circles a few times because of misunderstandings with the requirements. To verify co-existence and the effectiveness of our application, a proper modeling effort with multiple devices contending for the LAA carrier should be done. This type of modeling environment is quite difficult to develop, and would merit a study of its own.

REFERENCES

- [1] E. Dahlman, G. Mildh, S. Parkvall, J. Peisa, J. Sachs, Y. Selén, J. Sköld, 5G wireless access: requirements and realization, *IEEE Communications Magazine*, Vol. 52, No. 12, 2014, pp. 42-47.
- [2] H. J. Kwon, J. Jeon, A. Bhorkar, Q. Ye, H. Harada, Y. Jiang, L. Liu, S. Nagata, B. L. Ng, T. Novlan, J. Oh, W. Yi, Licensed-assisted access to unlicensed spectrum in LTE release 13, *IEEE Communications Magazine*, Vol. 55, No. 2, 2017, pp. 201-207.
- [3] 3GPP TR 36.889 V13.0.0, "Study on Licensed-Assisted Access to Unlicensed Spectrum," Jun. 2015.
- [4] LAA standardization: coexistence is the key, 3GPP, web page. Available (accessed 10.05.2017): http://www.3gpp.org/news-events/3gpp-news/1789-laa_update.
- [5] A. Mukherjee, J. F. Cheng, S. Falahati, H. Koorapaty, D. H. Kang, R. Karaki, L. Falconetti, D. Larsson, Licensed-assisted access LTE: coexistence with IEEE 802.11 and the evolution toward 5G, *IEEE Communications Magazine*, Vol. 54, No. 6, 2016, pp. 50-57.
- [6] 3GPP TS 36.213, v13.5.0, 3GPP, 2017, pp. 351-356.
- [7] F. Gebali, *Algorithms and Parallel Computing*, Wiley, Hoboken, 2011, pp. 365.
- [8] J.W. Cooley, J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, Vol. 19, No. 90, 1965, pp. 297-301.
- [9] H. Urkowitz, Energy detection of unknown deterministic signals, *Proceedings of the IEEE*, Vol. 55, No. 4, 1967, pp. 523-531.
- [10] M. Ohta, M. Taromaru, T. Fujii, O. Takyu, Spectrum edge decision method for energy detection with FFT, 2017 International Conference on Information Networking (ICOIN), pp. 271-276.
- [11] E. Auchard, Nokia, NTT DoCoMo prepare for 5G ahead of Tokyo Olympics launch, web page. Available (accessed May 15): <http://www.reuters.com/article/us-telecoms-mwc-ntt-docomo-idUSKBN0LY0FD20150302>.
- [12] R. Karaki, J. F. Cheng, E. Obregon, A. Mukherjee, D. H. Kang, S. Falahati, H. Koorapaty, O. Drugge, Uplink performance of enhanced licensed assisted access (eLAA) in unlicensed spectrum, 2017 IEEE Wireless Communications and Networking Conference (WCNC), pp. 1-6.

- [13] K. Keutzer, S. Malik, A. R. Newton, From ASIC to ASIP: the next design discontinuity, *Computer Design: VLSI in Computers and Processors*, 2002. Proceedings. 2002 IEEE International Conference on, pp. 84-90.
- [14] J. Henkel, Closing the SoC design gap, *Computer*, Vol. 36, No. 9, 2003, pp. 119-121.
- [15] TTA-Based Co-Design Environment, Tampere University of Technology and contributors, web page. Available (accessed 20.05.2017): <http://tce.cs.tut.fi/>.
- [16] ASIP Designer, Synopsys, Inc., web page. Available (accessed 20.05.2017): <https://www.synopsys.com/dw/ipdir.php?ds=asip-designer>.
- [17] J.L. Hennessy, D.A. Patterson, *Computer architecture : a quantitative approach*, 5th ed. Morgan Kaufmann, Waltham (MA), 2012, xxvii, pp. 493.
- [18] H. Corporaal, M. Arnold, Using transport triggered architectures for embedded processor design, *Integrated Computer-Aided Engineering*, Vol. 5(1998), No. 1, 1998, pp. 19-38.
- [19] V. Zivojnovic, S. Pees, H. Meyr, LISA-machine description language and generic machine model for HW/SW co-design, *VLSI Signal Processing*, IX, pp. 127-136.
- [20] LISA, ICE RWTH Aachen University, web page. Available (accessed 20.05.2017): <https://www.ice.rwth-aachen.de/research/tools-projects/lisa/lisa>.
- [21] A. Fauth, J. Van Praet, M. Freericks, Describing instruction set processors using nML, *Proceedings the European Design and Test Conference. ED&TC 1995*, pp. 503-507.
- [22] The LLVM Compiler Infrastructure Project, University of Illinois at Urbana-Champaign, web page. Available (accessed 23.05.2017): <http://llvm.org/>.
- [23] VCS, Synopsys, Inc, web page. Available (accessed 22.05.2017): <https://www.synopsys.com/verification/simulation/vcs.html>.
- [24] DC Ultra, Synopsys, Inc, web page. Available (accessed 22.05.2017): <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html>.
- [25] UVM Community, Accellera Systems Initiative, web page. Available (accessed 23.05.2017): <http://www.accellera.org/community/uvm/>.

